目 录

[1 项目需求简介 1](#_Toc134538459)

[1.1蔡徐坤和铁山靠背景简介 1](#_Toc134538460)

[1.2项目简介 2](#_Toc134538461)

[1.3项目意义 2](#_Toc134538462)

[2 项目数据模型说明 3](#_Toc134538463)

[2.1 部件基类模型 3](#_Toc134538464)

[2.2 关节，肢体基类模型 7](#_Toc134538465)

[2.3 机器人类模型 9](#_Toc134538466)

[2.4 其余类设计 12](#_Toc134538467)

[3 项目详细设计及成果展示 15](#_Toc134538468)

[3.1 困难关键点 15](#_Toc134538469)

[3.2 困难关键点实现 15](#_Toc134538470)

[3.2 1中分头曲面解决 15](#_Toc134538471)

[3.2 2 髋关节位置解决 17](#_Toc134538472)

[3.2 3手的绘制 18](#_Toc134538473)

[3.2 4 机器人自动运动的动画展示 19](#_Toc134538474)

[3.2 5 铁山靠系列动作实现 20](#_Toc134538475)

[3.2 6 铁山靠恢复到立正姿态 29](#_Toc134538476)

[3.2 7 实时暂停，旋转视角 33](#_Toc134538477)

[3.2 成果展示 36](#_Toc134538478)

[4 项目收获 41](#_Toc134538479)

[附录 42](#_Toc134538480)

# 1 项目需求简介

## 1.1蔡徐坤和铁山靠背景简介

蔡徐坤，俗称“鸡哥”，目前网上知名明星。之前本身作为娱乐圈从业人员名气较小，其出名的主要原因源于2018年在《偶像练习生》中的才艺表演。在2019年1月，蔡徐坤获任NBA新春贺岁形象大使，众多网友对此持嘲讽态度，挖出了他之前在《偶像练习生》中的一段才艺表演的视频。在视频中，蔡徐坤所展现的球技与形象大使完全不符，再加之配乐的第一句歌词“只因你太美”，因口音语速过快被读成了“鸡你太美”，充满滑稽。因该视频蔡徐坤在网上迅速遭到很多人的不满。很多人通过恶搞这段视频来讽刺形象大使甚至娱乐圈，此时蔡徐坤作为负面人物在网上爆红。后续蔡徐坤团队为此发律师函更加剧损害了本人在网上的形象，也使其一直保持了热度，没有随着时间推移热度下降。

之后，由于肖战等人的行迹更加恶劣，相比之下蔡徐坤基本没有损害他人利益的行迹，人们对蔡徐坤的态度逐渐缓和。再加上之前很多网友跟风不明缘由就贬低蔡徐坤，后来了解了原委后态度大有缓和。此时蔡徐坤逐渐由负面形象转为中立甚至正面形象，后来蔡徐坤本人也表示习惯了网友的恶搞。由于本身视频没有低俗内容，而且特征简单鲜明，蔡徐坤在2022年开始变得越来越火，基本上每个年轻人都知道他。篮球，鸡，律师函，背带裤等词已经成了蔡徐坤的代言词目前网上仍然有很多鬼畜的二创视频，但本身已不具有恶意，更多的是一种调侃，乐趣和创意成了相关二创的主旋律。

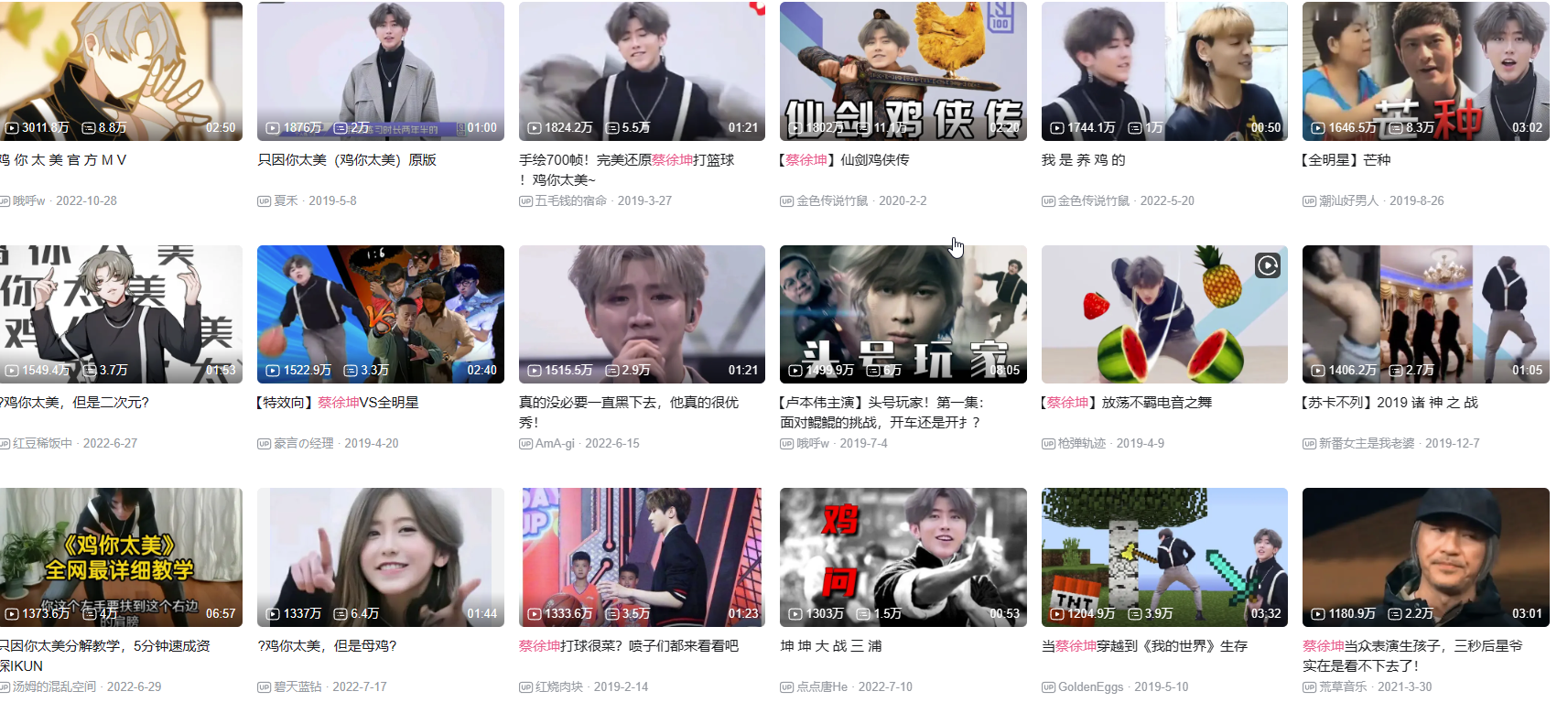


图1.1 蔡徐坤相关视频在b站的播放量

铁山靠，本指八极拳中的一种动作。由于蔡徐坤在才艺展示视频中的顶肩动作酷似铁山靠，同时顶肩又是舞蹈中的第一个动作，该动作也被称为“铁山靠”，随着蔡徐坤一起出名，成为了蔡徐坤的代名动作。网上也陆续出了很多铁山靠的动作教学。

铁山靠的大概动作：人立正后扎小马步，双手相握置于腹前，按照右上，左下，右下，左上的顺序移动身体（后续本文都称之为C型运动），顶肩，甩头。

## 1.2项目简介

本项目主要基于C++和opengl，运用了C++面向对象思想结合C++11的特性。本项目除了光源，渲染参考已有项目外(<https://blog.csdn.net/weixin_42584129/article/details/103213268>)，其余均为自己完成。项目搭建了机器人的类的框架，通过框架构建了一个机器人，机器人具有躯体，四肢和关节。。机器人实现了转向，走路，铁山靠以及从各个动作恢复到立正的动作。同时整个系统通过多线程实现了实时暂停，同时支持机器人在做运动时也能自由转动视角，全方位观察。

本项目中机器人的中心位置为躯体底面矩形的几何中心。x轴屏幕向右，y轴屏幕向上，z轴屏幕朝外，符合右手定则。机器人朝向z轴正方向。机器人的身体是最上层的部件，决定了其余所有的部件。

## 1.3项目意义

首先具有教学意义。鉴于蔡徐坤几乎人尽皆知，出于兴趣，追求乐趣或者打算在才艺展示上表演等原因存在有想学他舞蹈尤其铁山靠的人。本项目的机器人实现了人体运动的主要肢体，关节，同时能够暂停，全方位观察，能细致全面地进行教学。

然后具有社交意义。由于蔡徐坤强大的知名度，本身有大量鬼畜视频代表有很大乐趣，加之本人没大错却被全网黑了3年处于同情心理，现在人们普遍对其抱有正面态度，蔡徐坤某种程度上可以算是一个大众共有且容易畅谈的话题，是陌生人拉近彼此距离的一个好方法。蔡徐坤作为流量明星历经3年不衰甚至还越来越火，很多相关的梗已经侵入日常生活了，可以预见在相当一段时间内仍然是众所周知的，能保证作为社交工具的普遍性。铁山靠作为蔡徐坤的代表，动作简单又有一定的社交意义，具有较高的性价比。

最后具有娱乐商业意义。徐坤的知名度，本身具有的乐趣性，公众的态度共同构筑了庞大的市场。蔡徐坤相关创作的本意就是给大众带来乐趣和创意的，本项目也为日后他人相关的创作提供了一定创意。

# 2 项目数据模型说明

## 2.1 部件基类模型

将机器人视为许多部件组成，部件本身作为一个基类，可以拓展出各个具体的部件类，例如头，躯体等。机器人的运动由关节带动下面连接的肢体，因此还可以拓展出关节和肢体基类，再往下拓展为具体的肘关节，大腿等。

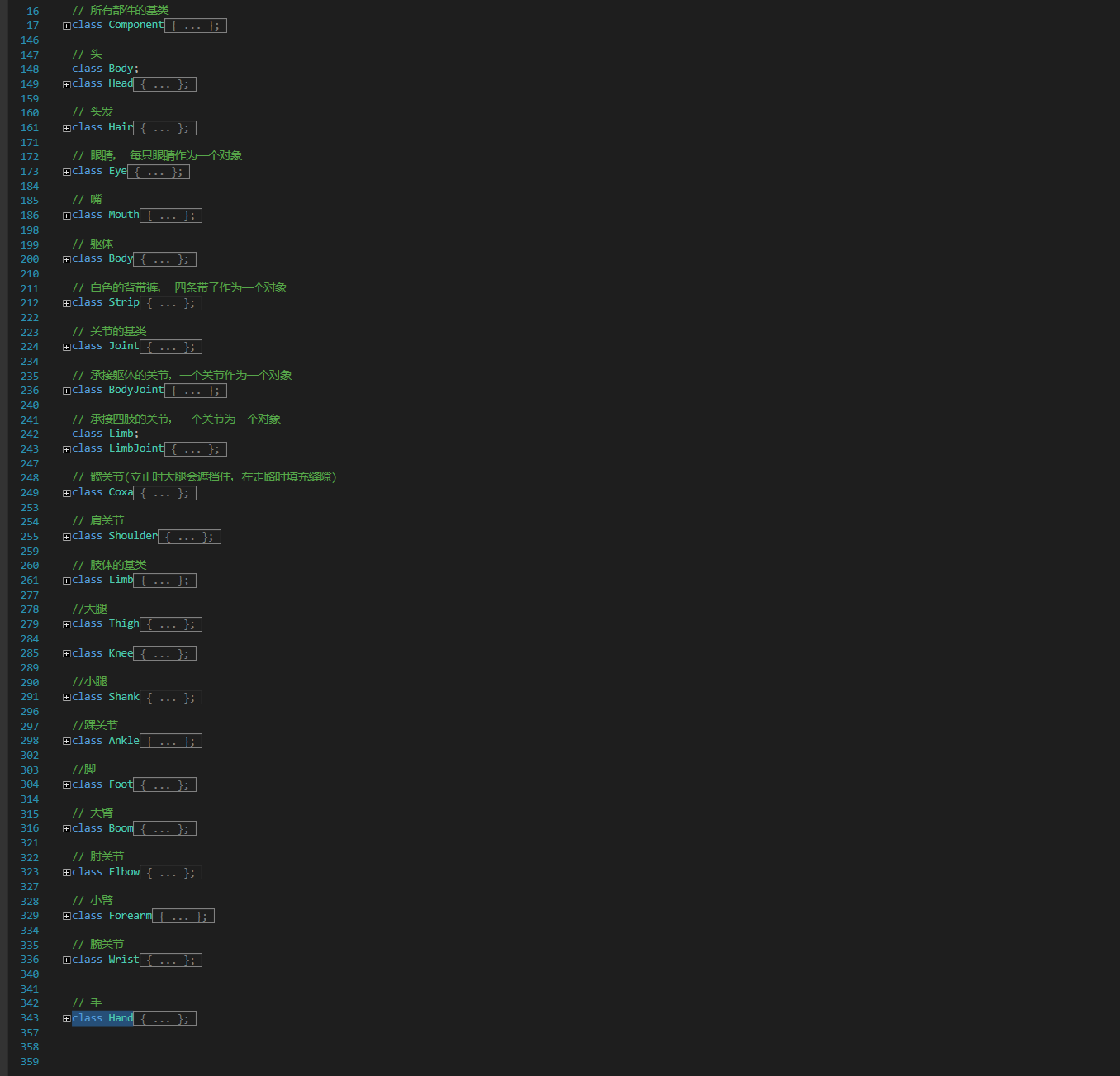


图2.1 各部件相关的类

对于每个部件，其基类蕴含的数据如下：

protected:

struct RotateStruct {

public:

float x = 0;

float y = 0;

float z = 0;

float angle = 0;

RotateStruct(float angle1, float x1, float y1, float z1) {

angle = angle1;

x = x1;

y = y1;

z = z1;

}

};

// 当前的

float x = 0; // x坐标

float y = 0; // y坐标

float z = 0; // z坐标

// 初始固定的

float initX = 0;

float initY = 0;

float initZ = 0;

float savedAngle = 0;

GLubyte color[3] = { 0, 0, 0 }; //颜色

vector<RotateStruct> RotateMemo; //记录各种旋转属性

float D2R(float angle) { return angle \* PI / 180; }

void InitCoordinate(); //处理opengl的坐标系到对应位置和角度

void Memo() { // 记录初始的坐标，用于精确回复

initX = x;

initX = y;

initZ = z;

}

std::map<string, Component\*> sufPart; // 有些小的部件依附于大的部件

unsigned int traceBack = -1; // 用于回溯到指定Rotate matrix

public:

const string label; // 标明这是哪一个部件

每个部件都可能有下游依附的部件，下游部件的大小位置等属性都受到该上游部件的影响，例如眼睛受到头的影响。sufPart存储该部件的下游部件。x, y, z坐标是对于它上游部件的相对坐标（默认最多只有一个上游部件），随着运动会变化。initX, initY, initZ的坐标是初始设立的相对坐标，不会变化，主要用于做完动作后恢复到初始状态，因float存储方式的问题，判断相等只能绝对值小于某个值，可能会不断积累误差导致误差越来越大，保存initX(Y)(Z)就可以规避这个问题。

RotateStruct, RotateMemo主要用于保存之前的旋转信息。平移可以叠加，不用存储，但是旋转的顺序不同，结果也不同，需要按顺序保存每个旋转的信息。其中RotateStruct为存储旋转信息的数据结构，保存旋转轴向量和旋转角度, RotateMemo为具体负责存储的变量。对于不同旋转轴的旋转不能叠加，但是对于相同旋转轴的旋转可以叠加。WholeRotate函数负责处理某个部件的旋转，这里就判断了旋转轴是否相同，相同则角度叠加，不同则分别存储旋转信息。由于机器人运动的动画需要通过循环每次在相同旋转轴上小幅改动角度并进行显示，如果在同一个旋转轴上旋转每次也保存一个新的旋转信息，那会有很多冗余的信息，导致大量时间和空间的浪费。更重要的是opengl是有栈深度限制的，大量冗余的信息会迅速占满深度，导致程序崩溃。

void Component::WholeRotate(float angle, float x, float y, float z, int accumulate) {

// 同方向连续轴的旋转可以叠加

if (RotateMemo.size() > 0) {

RotateStruct& last = RotateMemo.back();

if (x == last.x && y == last.y && z == last.z) {

last.angle += angle;

}

else {

RotateMemo.emplace\_back(angle, x, y, z);

RotateStruct tmp = RotateStruct(angle, x, y, z);

}

}

else {

RotateMemo.emplace\_back(angle, x, y, z );

}

}

这样有个弊端，就是相同旋转轴的角度可能会被覆盖，但是恢复动作需要未之前被覆盖的角度。这里通过savedAngle来实现，savedAngle用来保存之前未被覆盖的角度。TraceBack用来记录想要回溯到哪一次旋转，在其之后的都会被弹出。如果TrackBack < 0，会清空整个旋转。当遇到需要单独保存的旋转时，可以调用SaveRotateMatrix()来实现，此时traceBack记录的就是当前最后一次的旋转。

void SaveRotateMatrix() {

savedAngle = RotateMemo.back().angle;

if (RotateMemo.size() == 0) return;

traceBack = RotateMemo.size() - 1;

}

可以调用Trace2Saved()方法来回退到之前保存的旋转，用于做完动作后恢复到初始状态。

void Trace2Saved() {

if (traceBack == -1 || traceBack >= RotateMemo.size()) {

RotateMemo.clear();

return;

}

int m = RotateMemo.size() - 1 - traceBack;

while (m--) {

RotateMemo.pop\_back();

}

RotateMemo.back().angle = savedAngle;

}

最后color表示颜色，label标明这个部件是什么部件。

Component基类中的大多方法着重于获取，设置私有字段，其中比较特殊重要的方法除了上述介绍的SaveRotateMatrix(), Trace2Saved(), WholeRotate()，还有Display(), PopLastRotate()。其中Display()作为纯虚函数，因为具体的每个部件的Display都不同，而部件本身作为一个概念应当被设计成抽象类。每个部件都需要重写Display()，完成自己单独的显示工作。除了body，其余部件都采用相对位置，这就需要每个上游部件在Display()还要有对所有下游部件Display()的调用，形成类似多叉树的调用方式。PopLastRotate()是弹出RotateMemo最后一个存储的旋转，如果当前旋转不需要了，或者旋转角为0，就可以弹出这个旋转不断回退，用于动作的恢复。

最后还有重要的Initcoordinate()方法，它的作用就是将部件存储的平移信息（相对位置）和旋转信息(RotateMemo)映射到opengl的ModelView上。通过改变opengl ModelView下的matrix来实现图像的显示，运动。

## 2.2 关节，肢体基类模型

本项目中机器人的运动都是通过上游的关节旋转带动下游肢体的旋转。所有关节都认为是为球，肢体认为是圆柱体。关节包含髋关节，肩关节，膝关节，肘关节，踝关节，腕关节。肢体包含大腿，小腿，大臂，小臂。

关节的基类如下：

class Joint : public Component {

protected:

float radius = 0.15f;

int slice = 20;

public:

Joint(Component\* preJoint, string label1, GLubyte color[3], float r = 0.15f, int slice1 = 20);

void Display() override;

const float& GetR() const { return this->radius; }

};

其中slice表示切片，用于设置gluCylinders()的slices和stacks。Slice越高圆柱越圆。认为所有关节都是球，所以所有关节的Display()都相同，就是根据相对坐标，半径，精度和颜色画一个球。这时就可以直接在基类中定义显示操作了，不用单独为每一个子类配备同样的Display()了。

由于髋关节和肩关节直接和身体相连，其余关节上游是肢体，两者的相对位置参考不同，因此还需要将关节拓展为上游是躯体的关节BodyJoint和上游是肢体的LimbJoint。

BodyJoint的类声明如下：

// 承接躯体的关节，一个关节作为一个对象

class BodyJoint : public Joint {

public:

BodyJoint(Body\* body, string label1, GLubyte color[3], float r = 0.15f, int slice1 = 20);

};

LimbJoint的类声明如下：

// 承接四肢的关节，一个关节为一个对象

class Limb;

class LimbJoint : public Joint {

public:

LimbJoint(Limb\* limb, string label1, GLubyte color[3], float r = 0.15f, int slice1 = 20);

};

两者的唯一区别就是上游部件的类型不同。

对于肢体，所有肢体的上游部件都是关节，所以只需要一个肢体基类即可。肢体基类声明如下：

// 肢体的基类

class Limb : public Component{

protected:

float radiusTop = 0.15f;

float radiusBottom = 0.15f;

float height = 1.0f;

int accurate = 70;

int slice = 70;

public:

Limb(Joint\* precursor, string label1, GLubyte color[3],

float r = 0.15, float h = 1.0f, int accurate1 = 70, int slice1 = 20);

void Display() override;

const float& GetRTop() const { return this->radiusTop; }

const float& GetRBottom() const { return this->radiusBottom; }

const float& GetH() const { return this->height; }

};

radiusTop和radiusBottom表示圆柱上顶面和下顶面的圆的半径。Height是圆柱的高，就是肢体的长度。Accurate和slice分别表示两个顶面圆形的质量和整个圆柱的圆的质量，这两个应该是相等的。由于将所有肢体都认为是圆柱，所有肢体公用同一套Display()，就是根据相对位置，上下半径，高，精度和颜色画一个圆柱。可以将Display()放置到肢体的基类中。

## 2.3 机器人类模型

由于机器人的很多具体实现比较复杂且代码较长，实现在第三部分详细设计中介绍，这里只介绍机器人类的主体设计。设计如下：

class Robot{

private:

const static float PI;

int angleRange = 30;

map<string, Component\*> assembly; // 机器人的各个部件

int turn = 0; // 朝向的角度

int turnRate = 5; // 转身的速度

float forwardRate = 0.04; // 前进的速度

float armLiftLimit = 30; // 抬手的极限角度

float legLiftLimit = 45; //走路抬腿的极限角度

float legLiftRate = 3; // 走路抬腿的速度

float HorseStepRate = 5; //扎马步各部件旋转的速度

float HorseStepLimit = 45; // 扎马步大腿的极限角度

float SwingLimit = 10; //铁山靠身体角度的幅度

float SwingRate = 0.05; // 铁山靠的动作速度

float SwingDLimit = 0.1;

std::stack<float> HorseStepHeight; // 记录扎马步导致身体的下降的栈，用于还原

float ResetRate = 5; // 恢复成立正姿势各个部件还原的转动速度

// 相对坐标，只有躯体的是绝对坐标

float x = 0; // x坐标, 为机器人的躯体底部平面的中心位置

float y = 0; // y坐标

float z = 0; // z坐标

bool mutex = 0; //不能在铁山靠的时候走路

bool pause = 0; //机器人暂停了

bool stopSwing= 0; //这轮铁山靠完停止

std::mutex m;

std::unique\_lock<std::mutex> nothing; // wait需要一个锁，因为这里不是传统的互斥，用变量代替这个锁了

void AddComponet(const string& label, Component\* part); // 根据已有的部件添加部件

template<typename T>

void SetPartProperty(T\* arr, T x, T y, T z);

void SetPartRadius(float& r, float value);

void TurnFrontBack(bool toward = 1); // 使机器人正面或者背面朝向屏幕

void WalkStand(); // 从走路姿态回归到立正姿态

void HorseStep(); // 从立正姿态转为马步姿态

void ResetSwing(); // 从摇摆中回复到马步姿态

void ResetHorseStep(); // 从马步姿态回复到立正姿态

void Reset(); // 回复到立正姿态

void Swing();

void(\*windowDisplay)(); // 指向整个屏幕的display函数

std::condition\_variable cv; //用于处理暂停

public:

Robot(int newX = 0, int newY = 0, int newZ = 0);

~Robot();

const float& GetTurn() const { return this->turn; }

void SetWindowDisplay(void(\* displayFunc)());

// void SetKeyBoard(void(\*Keyboard)(unsigned char, int, int));

void SetLegLiftRate(float rate) { this->legLiftRate = rate; }

void SetLegLiftLimit(float limit) { this->legLiftLimit = limit; }

void SetTurnRate(int rate) { this->turnRate = rate; }

void SetForwardRate(float rate) { this->forwardRate = rate; }

void SetAngleRange(int range) { this->angleRange = range; }

void Generate(); // 生成各个需要的部件

void Display(); // 显示所有的部件

void ClockwiseRotate(); // 顺时针旋转

void AntiClockwiseRotate(); //逆时针旋转

void WalkForward(); //向前行走

void WalkBackward(); //向后走

void Dance(); // 模仿蔡徐坤的铁山靠

void WindowDisplay() { this->windowDisplay(); }

void Greeting(); // 问候语

// 铁山靠的时候要上锁，期间不能前后移动

void LockMyMutex() { mutex = 1; }

void UnlockMyMutex () { mutex = 0; }

bool IsLock() { return this->mutex; }

bool IsPause() { return this->pause; }

void ChangePauseState() {

if (pause) pause = 0;

else pause = 1;

}

void Notify() { this->cv.notify\_all(); }

void StopSwing() { this->stopSwing = 1; }

};

其中大部分字段都有注释了，简单的就直接跳过了。Mutex, unqiue\_lock, condition\_variable涉及到多线程，主要解决暂停和实时的视角旋转，会在第三部分详细设计中介绍，这里主要介绍和机器人运动相关的字段和操作。

Assembly，根据部件名称存储对应的部件。部件类本身实现了多态，所以对外可以全部以基类的形式存在，机器人不用对部件有深入的了解。

机器人的铁山靠动作作为一个整体，先背对屏幕立正，然后扎马步，接着侧身，之后C型摇摆，某个时刻停止，最后动作恢复。这些操作具有原子性，不可分割，因此TurnFrontBack(), HorseStep(), Swing(), Reset()这些动作都是私有字段，只有在Dance()中可以被调用，外部不能直接调用。其中TurnFrontBack()包含了转身和立正，爆 Swing()中包含了侧身和C型摇摆。Reset()中有ResetSwing()和ResetHorseStep(), 分别对应恢复到马步姿态和恢复到立正姿态。这些方法的具体实现会在详细设计中介绍。

对于void(\*windowDisplay)()函数指针，由于机器人的铁山靠是连续的动作，而且只有开始有一次的按键输入，不能像走路转身那样依赖glutKeyboardFunc()来依靠按键按住来实现动画效果。机器人要通过循环不断小幅调整角度，每次调角度进行显示来达到动画的效果，这就需要机器人有整个窗口的display()函数。由于不是Robot类内部的函数，不能直接调用，需要函数指针，通过函数指针获取到display()来进行调用。

最后是Generate()和Display()。Generate()用于生成所需的部件，指定初始机器人位置和身体大小，各部件的相对位置和颜色。Component::Display()以一种类似多叉树的方式连锁调用，每个部件的Display()会显示该部件及其所有下游部件，所以对于机器人本身，只要调用Body::Display()就可以了。

## 2.4 其余类设计

其余类为两个静态类：DrawTool和MathOperation，这两个都是工具类。

DrawTool主要提供了opengl没有现成的一些基本图形，本项目中是实心圆和圆柱体。实心圆就通过不断画多边形来实现。Opengl提供的gluCylinder()能画不封顶的圆柱。只需在两个顶面调用自己类中的DrawCircle()就可以封顶，从外观看上去与实心圆柱无异。需要注意的是DrawCircle()画的圆默认是在XOY平面的，而且gluCyliner()画的圆柱的轴心方向沿z轴正方向，正面在屏幕靠外的一边。而机器人要用到的圆柱的圆是在ZOX平面，正面朝y轴正方向，需要绕x轴逆时针旋转。旋转后对ModelView平移，平移到圆柱上表面圆的圆心处。默认情况下DrawTool::DrawCylinder()画的圆柱的母线沿y轴正方向。

DrawTool的两个绘图函数实现如下：

void DrawTool::DrawCircle(float R, int accurate)

{

glBegin(GL\_POLYGON);

for (int i = 0; i < accurate; i++)

glVertex3f(R \* cos(2 \* PI \* i / accurate), R \* sin(2 \* PI \* i/ accurate), 0);

glEnd();

}

void DrawTool::DrawCylinder(float Rtop, float Rbottom, float H, int accurate, int slice)

{

glPushMatrix();

glRotatef(-90, 1, 0, 0); //上面是外面

DrawCircle(Rtop, accurate);

gObj = gluNewQuadric();

glTranslatef(0, 0, -H / 1.0f);

gluQuadricDrawStyle(gObj, GLU\_FILL);

gluQuadricNormals(gObj, GLU\_SMOOTH);

gluCylinder(gObj, Rbottom, Rtop, H, slice, slice);

glRotatef(180, 1, 0, 0); // 下面是外面

DrawCircle(Rbottom, accurate);

glPopMatrix();

}

MathOperation主要提供了一些对于三维float向量的操作，其中最主要的是RotateAnyAxis()，作用是的得到任意向量绕过向量起点的任意轴按左手定则旋转任意角度后的向量。其中对应的旋转矩阵**R**如下:

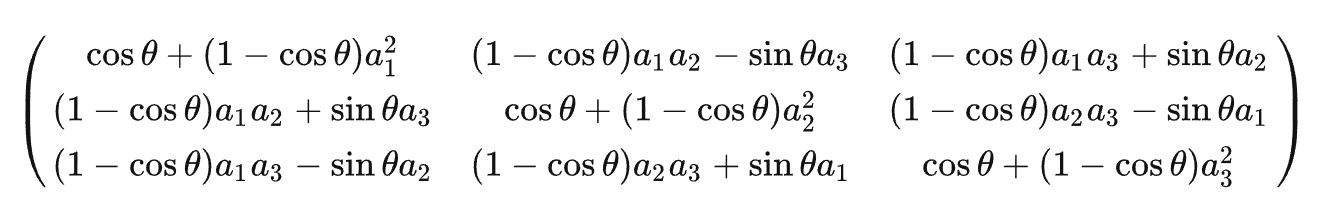


图2.2 任意向量绕过向量起点任意轴按左手旋转任意角度的旋转矩阵

其中θ为旋转角度, , , 为旋转轴方向上的单位向量。具体实现的代码如下：

void MathOperation::RotateAnyAxis(const float \*const pos, const float \*const axis, const float theta, float \* newPos) {

float rotateMatrix[3][3] = { {0, 0, 0}, {0, 0, 0}, {0, 0, 0} };

float cosTheta = cos(theta \* PI / 180.0f);

float sinTheta = sin(theta \* PI / 180.0f);

float x = axis[0], y = axis[1], z = axis[2];

float sum = sqrt(x \* x + y \* y + z \* z);

x = x / sum;

y = y / sum;

z = z / sum;

rotateMatrix[0][0] = x \* x \* (1 - cosTheta) + cosTheta;

rotateMatrix[0][1] = x \* y \* (1 - cosTheta) - z \* sinTheta;

rotateMatrix[0][2] = x \* z \* (1 - cosTheta) + y \* sinTheta;

rotateMatrix[1][0] = x \* y \* (1 - cosTheta) + z \* sinTheta;

rotateMatrix[1][1] = y \* y \* (1 - cosTheta) + cosTheta;

rotateMatrix[1][2] = y \* z \* (1 - cosTheta) - x \* sinTheta;

rotateMatrix[2][0] = x \* z \* (1 - cosTheta) - y \* sinTheta;

rotateMatrix[2][1] = y \* z \* (1 - cosTheta) + x \* sinTheta;

rotateMatrix[2][2] = z \* z \* (1 - cosTheta) + cosTheta;

newPos[0] = newPos[1] = newPos[2] = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

newPos[i] += rotateMatrix[i][j] \* pos[j];

}

}

}

通过这个方法可以简单模拟ModelView的运动。在本项目中，该方法主要用于计算扎马步身体下降的距离。

# 3 项目详细设计及成果展示

## 3.1 困难关键点

这里只列举实施中碰到的困难且关键性的地方，具体解决措施在下一章节有详细介绍。

1. 机器人构建中，蔡徐坤的中分头的处理，不是一个简单的平面，是一个曲面。
2. 机器人构建中，髋关节的处理。如果没有髋关节，身体直接与大腿相连，会导致在运动中大腿与身体产生缝隙。如果有髋关节，髋关节的位置应置于何处，髋关节不同肩关节位于于躯体边缘，不能简单置于躯体下方与躯体下表面相切。髋关节有点内嵌在躯体中。
3. 机器人构建中，手的处理。手不像脚可以用一个扁平的立方体代替。如果要画手，手有五个手指，精细度和复杂度以我目前的水平很难达到
4. 机器人运动中，铁山靠作为一个自动的动画，不同于走路，转向通过glutKeyboardFunc()的按键来驱动展示运动过程。移动，旋转好后通过RobotIkun.cpp中的display()只能显示运动完的过程。
5. 机器人运动中，铁山靠一系列动作：马步，侧身，C型摇摆运动的处理。涉及到许多关节而且是在三维空间的转向，精密计算对数学的要求特别高。C型摇摆的可重复性，每一轮的C型摇摆结束时的位置，旋转信息相同。
6. 如何从铁山靠的动作中恢复到立正姿态，因为铁山靠需要很多的运动，会产生大量的平移和旋转。
7. 如何在机器人自动运动的过程中随时暂停或者旋转视角。因为自动运动是在一套循环体系里，此时程序无法循环glutMainLoop()中，无法响应此时的按键输入。

## 3.2 困难关键点实现

### 3.2 1中分头曲面解决

左右对称，先只考虑x > 0的情况。中分头采用了余弦函数来模拟。采用微积分的思想，通过不断构造余弦函数曲线来构造曲面。首先需要构造z = 0平面处的余弦曲线。这里只有一个点的限制，(body->scale[0] / 2, 0)，显然是无法确定三角函数的。这里为了计算简便，将余弦函数的上下左右的平移量设为0，认为在x = 0时取到最大值，在x = body->scale[1]时第一次取到0，为T / 4。这样body->scale[0]就是T / 2, 可以根据周期公式得到ω = PI / body->scale[0]。最后的振幅A给定初值，为0.15。此时的余弦函数f = 0.15 \* cos(PI / body->scale[0] \* x)。

Z = 0平面处的曲线构造好后，只需要沿z轴不断平移复制，即可构造一个曲面。但这存在一个问题，在[-body->scale[0], body->scale[0]]，区间内，头发应该要延伸到头顶，但此时改区间内的头发是腾空的，因此还需要在余弦函数和头顶这块区域进行填充，但要注意当曲线在头顶下方时就不填充了。

整体的实现代码如下：

class Hair : public Component{

private:

float density = 0.01; //循环的步长

float lineWidth = 0.5;

Head\* head = nullptr; //依附在哪个头对象上

public:

Hair(Head\* head, string label1, GLubyte color[3], float density1 = 0.01f, float linewidtdh1 = 0.5f);

void Display() override;

};

void Hair::Display()

{

glPushMatrix();

InitCoordinate();

glLineWidth(this->lineWidth);

const GLfloat length = this->head->getScale()[2] / 2;

const GLfloat width = this->head->getScale()[0];

for (GLfloat j = -length; j <= length; j += density) {

glBegin(GL\_LINE\_STRIP);

for (GLfloat i = -width; i <= width; i += density) {

GLfloat height = 0.15 \* cos(PI / width \* i);

for (GLfloat h = 0; h <= height; h += density) {

glVertex3f(i, h, j);

}

glVertex3f(i, height, j);

}

glEnd();

}

glLineWidth(1);

glPopMatrix();

}

### 3.2 2 髋关节位置解决

髋关节的上游部件是身体，因此是BodyJoint，派生自BodyJoint。

此时明确的是：髋关节是肯定需要的，否则大腿运动与身体会产生大量缝隙同时不能整个关节外置在身体中，这样不符合人体外观。髋关节一部分要内置在身体中不能大从身体中凸出来，同时也不能太小导致大腿很小，这就对髋关节的半径有严格的限制。最后的方案是髋关节上半球藏于身体中，下半球在立正情况下与大腿重合。这样尽可能保证整体符合人体外观的同时也能填充缝隙。

对于髋关节的半径，考虑到人身体不会z轴很小，x,y轴很大或者z轴很大，x,y轴很小，人的大腿基本与身体在z轴方向的长度相同，最终决定髋关节的半径恒等于body->scale[2] / 2，没有进行特判。

综上，髋关节的构造方式如下：

class Coxa : public BodyJoint {

public:

Coxa(Body\* body, string label1, GLubyte color[3], int num, float r = 0.15f ,int slice1 = 20);

};

Coxa::Coxa(Body\* body, string label1, GLubyte color[3], int num, float r, int slice1) :

BodyJoint(body, label1, color, r, slice1)

{

if (body == nullptr) return;

this->x = - num \* body->getScale()[0] / 2.0f + num \* this->radius / 1.0f;

this->y = - body->getScale()[1] / 2.0f;

this->z = 0;

void ChangeAngles(int variation[3]);

body->AddSufPart(this);

Memo();

}

其中，num区分是左髋节还是右髋关节。

### 3.2 3手的绘制

通过观察，发现手的整体偏向于椭圆，可以用椭圆来代替手。Opengl中glScalef()可以设置x:y:z，通过改变比例就可以将圆改造为椭圆。圆已经在DrawTool类中有封装好的方法， 可以直接使用。

但此时会有一个问题，手是厚度的，不能有用一个平面来代替。椭圆沿x轴方向延伸，可以形成一个椭圆圆柱。之前DrawTool::DrawCylinder()的默认方向与此处不相符，需要沿z轴旋转90度。

手的实现代码如下：

class Hand : public Component {

private:

Wrist\* wrist = nullptr;

float ratio = 1.0f; // x和z轴的比例，x轴方向为1

float radius = 0.15f;

float height = 0.1f;

int accurate = 70;

int slice = 20;

public:

Hand(Wrist\* wrist, string label1, GLubyte color[3], float r = 0.15f, float h = 0.1f,

float ratio1 = 1.0f, int accurate1 = 70, int slice1 = 20);

void Display() override;

};

void Hand::Display()

{

glPushMatrix();

InitCoordinate();

glRotatef(90, 0, 0, 1);

glTranslatef(0.0f, this->height / 2.0f, 0.0f);

glScalef(1.0, 1.0f, this->ratio);

DrawTool::DrawCylinder(this->radius, this->radius, this->height, this->accurate, this->slice);

glPopMatrix();

}

### 3.2 4 机器人自动运动的动画展示

为了展示机器人的运动过程，每次运动的幅度要小，同时每次运动需要重新绘制整个窗口，这就需要Robot类中有能够重新绘制整个窗口的方法。比较直接的方法是在Robot类中重新写一个成员方法，其内容和RobotIkun.cpp中的display()方法相同，不过这样会导致大量的冗余，而且不易更改，需要同时维护两个display()函数。从逻辑上讲，Robot类中重新绘图的函数应该和RobotIkun.cpp中的display()完全一样，但是Robot类无法看到RobotIkun.cpp中的内容，无法直接通过函数名调用。这种情况就需要函数指针了，Robot类中声明一个函数指针，然后提供对外的方法，这样在RobotIkun.cpp中就可以调用相应的方法来更改Robot中的函数指针，使其指向display()函数。这样Robot就有了display()函数了。

之后在机器人自动运动的时候，设定好最终运动的状态，然后通过循环，每次小幅度调整并且重绘窗口，就可以完成机器人自动动作的动画了。

具体相关的实现如下:

在Robot类中对应的成员:

void(\*windowDisplay)(); // 指向整个屏幕的display函数

void SetWindowDisplay(void(\* displayFunc)());

void WindowDisplay() { this->windowDisplay(); }

在RobotIkun.cpp中的调用：

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

glutInitWindowSize(800, 600);

glutInitWindowPosition(100, 100);

glutCreateWindow("Robot");

std::cout << "Please Enjoy the Robot" << std::endl;

SetupRC();

robot->Generate();

robot->SetWindowDisplay(display);

glutDisplayFunc(display);

以机器人转向背面的动作为例：

void Robot::TurnFrontBack(bool toward) {

if (this->turn <= 180) {

while (this->turn <= 180) {

cv.wait(nothing, [this] {return !this->IsPause(); });

if (toward) this->ClockwiseRotate();

else this->AntiClockwiseRotate();

this->WindowDisplay();

Sleep(10);

}

turn = 180;

}

else {

while (this->turn >= 180) {

cv.wait(nothing, [this] {return !this->IsPause(); });

// cv.wait(nothing, [this] {return !this->IsPause(); });

if (toward) this->AntiClockwiseRotate();

else this->ClockwiseRotate();

this->WindowDisplay();

Sleep(10);

}

turn = 180;

}

}

其中while()条件中决定了机器人运动结束后的状态，循环体中每次只运动一小点，然后通过this->WindowDisplay()来重绘窗口，这样整体上就显示了机器人的运动过程，而不是只显示最终运动好的结果。

### 3.2 5 铁山靠系列动作实现

铁山靠的动作主要流程为：

1. 转身背对屏幕
2. 从当前走路姿态恢复到立正姿态
3. 蹲马步，抬手臂
4. 身体往右倾斜
5. 身体依次朝右上，左下，右下，左上运动，并且重复该动作

对于1，通过TurnFrontBack()方法来实现，根据当前的朝向判断顺时针还是逆时针旋转，旋转过程通过ClocwiseRotate()和AntiClockwiseRotate()实现，期间通过自动动画的形式展现运动过程。后续的动作都是自动进行的，都是采用类似的while循环小幅度调整后绘制的结构，在之后的流程中将不再赘述。

最终为了防止角度变化可能不能正好到180，最终需要强制赋值为180。

void Robot::TurnFrontBack(bool toward) {

if (this->turn <= 180) {

while (this->turn <= 180) {

cv.wait(nothing, [this] {return !this->IsPause(); });

if (toward) this->ClockwiseRotate();

else this->AntiClockwiseRotate();

this->WindowDisplay();

Sleep(10);

}

turn = 180;

}

else {

while (this->turn >= 180) {

cv.wait(nothing, [this] {return !this->IsPause(); });

// cv.wait(nothing, [this] {return !this->IsPause(); });

if (toward) this->AntiClockwiseRotate();

else this->ClockwiseRotate();

this->WindowDisplay();

Sleep(10);

}

turn = 180;

}

}

对于2，通过WalkStand()来实现。思想就是从当前姿态往后走，如果左大腿的角度与之前不同，那么说明此时经过了立正的姿态，但可能因为某些原因，可能没有正好处于立正姿态，因此最后需要清空所有的旋转，绝对处于立正姿态。因为traceBack此时为默认值-1，Trace2Saved()作用相当于清空所有的旋转信息。具体的代码如下：

void Robot::WalkStand()

{

auto leftCoxa = this->assembly["LeftCoxa"];

auto rightCoxa = this->assembly["RightCoxa"];

auto leftShoulder = this->assembly["LeftShoulder"];

auto rightShoulder = this->assembly["RightShoulder"];

float orignAngle = leftCoxa->GetAngle();

while (orignAngle \* leftCoxa->GetAngle() > 0) {

cv.wait(nothing, [this] {return !this->IsPause(); });

this->WalkBackward();

this->WindowDisplay();

Sleep(20);

}

rightCoxa->Trace2Saved();

leftCoxa->Trace2Saved();

leftShoulder->Trace2Saved();

rightShoulder->Trace2Saved();

this->WindowDisplay();

Sleep(20);

}

对于3，从这里开始，运动就变得复杂了，出现了三维空间上的连续旋转了。在此之前，几乎所有的量都是采用变量，通过变量关系来改变的，易于修改，改变规模。但是从这里开始，再采用变量，通过找变量关系构建关系式的话，会涉及到非常复杂的数学计算，这也超出了本人的数学范围，也与opengl运动关系设计的关系不大。因此，从该步骤往后，都是采用数字，通过感官不断调参来实现对应的动作。

蹲马步大腿往外斜，小腿往里斜，身体往前倾，人重心往下降。同时抬手臂，大臂往外斜，小臂往里斜，手大概置于腹前。这里每个部件的旋转信息都是尝试出来的，没有解释性。不过对于重心的下降，还是通过一定的计算的。

这里重心的下降的原因在于大腿和小腿的弯曲程度。对于小腿，是在大腿旋转基础上二次旋转，情况过于复杂，而且通过观察发现，本项目中小腿于y轴的角度不大，几乎不影响重心的降低，固不考虑小腿弯曲带来的重心降低，只考虑大腿的。对于大腿，可以设想大腿的根部为向量起点，大腿长度为向量长度，设为h，设该向量为posThigh[3]，那么由于之前的步骤保证了此时是绝对立正姿态，故初始的posThigh = {0, -h, 0}。由旋转轴和旋转角度已知，通过MathOperation::RotateAnyAxis()可以计算出posThigh第一次旋转后的向量newPos。通过posThigh[1]和newPos[1]的差就可以得到这一次旋转导致的高度较低。之后更新posThigh为newPos，进行下一次的旋转，同样的方法计算高度差，直到循环结束。

其中需要注意的是，在后续动作的恢复中，仍然需要这些高度差用来恢复重心，只不过是反过来的。可以在蹲马步的时候就将存储好这些高度差，这样在恢复的时候就不用再计算一遍了。由于是后进先出，可以采用栈来存储，栈记为HorseStepHeight。

具体代码如下：

void Robot::HorseStep()

{

auto leftShoulder = this->assembly["LeftShoulder"];

auto rightShoulder = this->assembly["RightShoulder"];

auto leftElbow = this->assembly["LeftElbow"];

auto rightElbow = this->assembly["RightElbow"];

auto leftCoxa = this->assembly["LeftCoxa"];

Thigh\* leftThigh = (Thigh\*)this->assembly["LeftThigh"];

auto rightCoxa = this->assembly["RightCoxa"];

Knee\* leftKnee = (Knee\*)this->assembly["LeftKnee"];

auto rightKnee = this->assembly["RightKnee"];

auto body = this->assembly["Body"];

// 这里只算上了大腿弯曲带来的下降，小腿的情况太复杂了

float posThigh[3] = { 0, -leftThigh->GetH(), 0 };

float newPosThigh[3] = { 0, 0, 0 };

while (rightCoxa->GetAngle() <= this->HorseStepLimit) {

cv.wait(nothing, [this] {return !this->IsPause(); });

leftShoulder->WholeRotate(-this->HorseStepRate \* 0.5, 1, 0.5, -1, 1);

leftElbow->WholeRotate(-this->HorseStepRate \* 3, 1, 2, 0, 1);

rightShoulder->WholeRotate(this->HorseStepRate \* 0.5, -1, 0.5, -1, 1);

rightElbow->WholeRotate(this->HorseStepRate \* 3, -1, 2, 0, 1);

rightCoxa->WholeRotate(this->HorseStepRate, -1, 0.5, -1, 1);

leftCoxa->WholeRotate(-this->HorseStepRate, 1, 0.5, -1, 1);

float axis[3] = { 1, 0.5 , -1 };

MathOperation::RotateAnyAxis(posThigh, axis, -this->HorseStepRate, newPosThigh);

body->SetY(body->GetY() + posThigh[1] - newPosThigh[1]);

this->HorseStepHeight.push(newPosThigh[1] - posThigh[1]); //压栈用于恢复

posThigh[0] = newPosThigh[0];

posThigh[1] = newPosThigh[1];

posThigh[2] = newPosThigh[2];

rightKnee->WholeRotate(-this->HorseStepRate, -1, 0.5, -1, 1);

leftKnee->WholeRotate(this->HorseStepRate, 1, 0.5, -1, 1);

body->WholeRotate(this->HorseStepRate / 3, 1, 0, 0, 1);

this->WindowDisplay();

Sleep(20);

}

}

4和5放在了一个方法中Swing()实现，这里也放在一起讲。同样，这里所有的旋转信息都是凭感觉调参得到的，没有可解释性。

侧身需要涉及到身体的两次不同的转动，先绕x轴转，在此基础上绕z轴转，旋转轴不同。这里通过Component::RotateMemo存储了两次旋转信息，通过InitCoordinate()作用到ModelView的matrix上。

侧身后的位置是一个特殊的位置，记为，不是C型运动的四个端点。初始先往右上运动，记为，位置相当于每次C型运动的起点，每轮运动都需要保证是从开始而且各个部件的旋转信息都相同，这样才能保证C型运动不会因为连续运动导致位置，角度不断偏移。保存的向量为originVector[3]来控制位置相同。Component::SaveRotateMatrix()来存储当前对应部件的旋转信息。因为后续的有些旋转是在同一个轴上会覆盖当前的角度，所以这里还需要保存角度。

从开始往左下运动，到达。从往右下运动到达。从开始往左上运动，这里需要回到。这里通过保证循环总次数，和身体的平移向量相反来保证的。两次的运动最终旋转的角度不同，不能用相同的while()条件，但能通过最终的旋转角度和每次旋转的步长的比例相同来保证同样的循环次数。

最后一步是从往右上运动回到。这里不再以最终旋转角度作为循环种植条件，而是body->position与originVector的距离作为终止条件。通过可以得到平移的向量，然后每次按一定步长来平移。由于计算精度的问题，不能保证最终与完全重合，只能根据步长控制在一定范围内。为了动画整体速度差不多，步长不能设置得太小，这就导致最终的误差可能较大。对于C型运动，x轴上的一定偏移是可以接受的，人自己做也可能有一点细微的差别，但是y轴因为地面不变，人不会上升或者下降，所以y轴上的偏移是不能接受的，这里通过控制y轴的距离，回到。回到还要复原角度，这里通过Trace2Saved()，回退到之前保存的旋转信息，顺便把存储的用不到旋转信息弹出。由于当前的旋转和Trace2Saved中读到的旋转可能差距过大，会导致机器人一下出现了瞬动，导致动作不连贯，这是目前项目还未能解决的问题。 因为opengl ModelView的matrix本身就有深度限制，所以要时刻弹出不用的旋转信息。

本项目中没有干预的情况下铁山靠将一直重复C型摇摆，按下R后停止，并恢复到初始状态。

侧身，C型运动代码如下：

// 受限于数学物理知识，没有经过计算，只是凭感觉调参, 动作可能不标准

void Robot::Swing() {

auto leftCoxa = this->assembly["LeftCoxa"];

auto rightCoxa = this->assembly["RightCoxa"];

auto leftKnee = this->assembly["LeftKnee"];

auto rightKnee = this->assembly["RightKnee"];

auto body = this->assembly["Body"];

auto rightElbow = this->assembly["RightElbow"];

Sleep(50);

bool isFirst = 1;

// 侧身

while (isFirst || abs(body->GetAngle()) <= this->SwingLimit) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

body->WholeRotate(1, 0, 0, 1, 1); //二次转动

rightCoxa->WholeRotate(-0.8, 0, 0, 1, 1);

leftCoxa->WholeRotate(-1.5, 0, 0, 1, 1);

leftKnee->WholeRotate(2, -2, 1, 0, 1);

this->WindowDisplay();

Sleep(20);

}

Sleep(50);

// 初始状态下往右上顶

isFirst = 1;

float moveVector[3] = { 0, 0, 0 };

float originVector[3] = { 0, 0, 0 };

while (isFirst || abs(leftCoxa->GetAngle()) <= this->SwingLimit) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

leftCoxa->WholeRotate(-3, 0, 1, 1, 1);

leftKnee->WholeRotate(-this->HorseStepRate, 1, 0.5, -1, 1);

rightCoxa->WholeRotate(5, 0, 1, 1, 1);

rightKnee->WholeRotate(this->HorseStepRate, -1, 0.5, -1, 1);

body->Translate(-0.03, 0.05, 0);

this->WindowDisplay();

Sleep(20);

}

originVector[0] = body->GetX();

originVector[1] = body->GetY();

originVector[2] = body->GetZ();

Sleep(50);

// 定为基准矩阵，每次右上角顶后需要同步到这些旋转矩阵

leftCoxa->SaveRotateMatrix();

rightCoxa->SaveRotateMatrix();

leftKnee->SaveRotateMatrix();

rightKnee->SaveRotateMatrix();

// int m = 2;

while (1) {

// 往左下回

isFirst = 1;

while (isFirst || abs(leftCoxa->GetAngle()) <= this->SwingLimit) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

leftCoxa->WholeRotate(3, 0, 1, 1, 1);

leftKnee->WholeRotate(this->HorseStepRate, 1, 0.5, -1, 1);

rightCoxa->WholeRotate(-5, 0, 1, 1, 1);

rightKnee->WholeRotate(-this->HorseStepRate, -1, 0.5, -1, 1);

body->Translate(0.05, -0.06, 0);

this->WindowDisplay();

Sleep(20);

}

Sleep(50);

// 往右下顶

isFirst = 1;

while (isFirst || abs(leftCoxa->GetAngle()) <= 3 \* this->SwingLimit) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

leftCoxa->WholeRotate(3.5, 0.5, 1, 0, 1);

leftKnee->WholeRotate(-this->HorseStepRate \* 1.4, 1, 0.5, -1, 1);

rightCoxa->WholeRotate(-1.5, 0, 0, 1, 1);

rightKnee->WholeRotate(this->HorseStepRate \* 0.3, -1, 0.5, -1, 1);

body->Translate(-0.06, -0.02, 0);

this->WindowDisplay();

Sleep(20);

}

Sleep(50);

// 往左上顶

isFirst = 1;

while (isFirst || abs(leftCoxa->GetAngle()) <= this->SwingLimit) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

leftCoxa->WholeRotate(-3.5f / 3, 0.5, 0, -3, 1);

leftKnee->WholeRotate(this->HorseStepRate \* 0.5, 1, 0.5, -1, 1);

rightCoxa->WholeRotate(2, 0, 0, 1, 1);

//rightKnee->WholeRotate(-\* this->HorseStepRate , -1, 0.5, -1, 1);

body->Translate(0.06, 0.02, 0);

this->WindowDisplay();

Sleep(20);

}

// 往右上顶, body和各个部件的旋转矩阵要与上一次右上顶的相同，保证整套动作能一直重复下去

isFirst = 1;

moveVector[0] = originVector[0] - body->GetX();

moveVector[1] = originVector[1] - body->GetY();

moveVector[2] = originVector[2] - body->GetZ();

float step = 0.1;

while (1) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

leftCoxa->WholeRotate(-4, -1, 0, 2, 1);

leftKnee->WholeRotate(-this->HorseStepRate \* 0.5, 1, 0.5, -1, 1);

rightCoxa->WholeRotate(3, 1, 0, 1, 1);

rightKnee->WholeRotate(this->HorseStepRate \* 0.5, -1, 0.5, -1, 1);

body->Translate(moveVector[0] \* step, moveVector[1] \* step, moveVector[2] \* step);

if (abs(originVector[1] - body->GetY()) <= 0.1) break;

this->WindowDisplay();

Sleep(20);

}

leftCoxa->Trace2Saved();

rightCoxa->Trace2Saved();

leftKnee->Trace2Saved();

rightKnee->Trace2Saved();

this->WindowDisplay();

if (this->stopSwing) {

this->stopSwing = 0;

return;

}

}

}

自此，机器人就已经能做整一套的铁山靠动作了。

### 3.2 6 铁山靠恢复到立正姿态

恢复动作发生在C型摇摆的右上角的位置。恢复动作与铁山靠的动作顺序相反，整套流程如下：

1. C型摇摆从复原到侧身动作
2. 从侧身恢复到马步动作
3. 从马步动作恢复到立正姿态

1和2在一个方法ResetSwing()中，这里也一直分析。此时位于P\_1，根据之前铁山靠的分析，此时每个部件没有冗余的旋转信息，即RotateMemo.back()就代表的是从侧身运动到P\_1的旋转信息。通过Component.GetRotateX/Y/Z()，可以获得此时的旋转轴。然后旋转角度为RotateMemo.back()的相反数即可。需要注意的是，恢复时的次序要颠倒，铁山靠中先平移，旋转的部件在恢复过程中后操作。

平移的恢复直接负向量就能解决。旋转的恢复分析如下：如果侧身的旋转轴与马步的旋转不同，那么恢复到侧身时RotateMemo.back().angle应该为0，因为旋转轴相同，旋转了相反度数的角。此时旋转是冗余的，直接弹出来获取上一层的旋转信息，用于后续的动作恢复。如果侧身的旋转轴与马步的旋转不同，那么恢复到侧身时RotateMemo.back().angle不为0，角度变为了之前旋转的角度，此时不应该弹出该旋转信息，此时的RotateMemo.back()自动就代表了上一个的旋转信息。

由于float计算方式的问题，旋转相反角度后当前的旋转角可能是一个非常小但不为0的数字，不能直接用等于0判断，需要绝对值小于一个范围。

侧身动作恢复到扎马步的过程与C型摇摆恢复到侧身类似，这里就不再进行分析

ResetSwing()具体的代码如下：

void Robot::ResetSwing() {

bool isFirst = 1;

auto leftCoxa = this->assembly["LeftCoxa"];

auto rightCoxa = this->assembly["RightCoxa"];

auto leftKnee = this->assembly["LeftKnee"];

auto rightKnee = this->assembly["RightKnee"];

auto body = this->assembly["Body"];

auto rightElbow = this->assembly["RightElbow"];

// 回复到侧身姿态

while (isFirst || abs(leftCoxa->GetAngle()) >= 0.0001) {

cv.wait(nothing, [this] {return !this->IsPause(); });

body->Translate(0.03, -0.05, 0);

isFirst = 0;

rightKnee->WholeRotate(-this->HorseStepRate, rightKnee->GetRotateX(), rightKnee->GetRotateY(), rightKnee->GetRotateZ(), 1);

rightCoxa->WholeRotate(-5, rightCoxa->GetRotateX(), rightCoxa->GetRotateY(), rightCoxa->GetRotateZ(), 1);

leftKnee->WholeRotate(this->HorseStepRate, leftKnee->GetRotateX(), leftKnee->GetRotateY(), leftKnee->GetRotateZ(), 1);

leftCoxa->WholeRotate(3, leftCoxa->GetRotateX(), leftCoxa->GetRotateY(), leftCoxa->GetRotateZ(), 1);

this->WindowDisplay();

Sleep(20);

}

Component\* container[5] = { leftCoxa, leftKnee, rightCoxa, rightKnee, body };

for (Component\* i : container) {

while (abs(i->GetAngle()) <= 0.01) i->PopLastRotate();

}

Sleep(50);

// 回复到扎马步姿态

isFirst = 1;

while (isFirst || abs(body->GetAngle()) >= 0.0001) {

cv.wait(nothing, [this] {return !this->IsPause(); });

isFirst = 0;

leftKnee->WholeRotate(-2, leftKnee->GetRotateX(), leftKnee->GetRotateY(), leftKnee->GetRotateZ(), 1);

leftCoxa->WholeRotate(1.5, leftCoxa->GetRotateX(), leftCoxa->GetRotateY(), leftCoxa->GetRotateZ(), 1);

rightCoxa->WholeRotate(0.8, rightCoxa->GetRotateX(), rightCoxa->GetRotateY(), rightCoxa->GetRotateZ(), 1);

body->WholeRotate(-1, body->GetRotateX(), body->GetRotateY(), body->GetRotateZ(), 1);

this->WindowDisplay();

Sleep(20);

}

for (Component\* i : container) {

while (abs(i->GetAngle()) <= 0.01) i->PopLastRotate();

}

Sleep(50);

}

对于3，旋转的恢复与之前类似，不过这里的平移恢复不是负向量，是通过扎马步HorseStep()中存储的HorseStepHeight栈来获取平移量的。每次循环中HorerStepHeight的栈顶信息就是之前扎马步时重心下降的距离，恢复时负向平移即可。不同于与之前的恢复，扎马步立正时候清除了所有旋转信息，此时RotateMemo中只有扎马步对应的旋转信息，不存在之前旋转与此时旋转有相同旋转轴的情况，恢复后的旋转角肯定都为0，可以不加判断直接弹出。

由于可能存在计算精度的误差，恢复后的重心可能与之前有所差距，所以最后根据initX/Y/Z，强制恢复到初始的位置。

具体的代码如下：

void Robot::ResetHorseStep()

{

auto leftCoxa = this->assembly["LeftCoxa"];

auto rightCoxa = this->assembly["RightCoxa"];

auto leftKnee = this->assembly["LeftKnee"];

auto rightKnee = this->assembly["RightKnee"];

auto body = this->assembly["Body"];

auto leftShoulder = this->assembly["LeftShoulder"];

auto rightShoulder = this->assembly["RightShoulder"];

auto leftElbow = this->assembly["LeftElbow"];

auto rightElbow = this->assembly["RightElbow"];

//std::cout << rightCoxa->GetRotateX() << " " << rightCoxa->GetRotateY() << " " << rightCoxa->GetRotateZ() << " " << rightCoxa->GetAngle() << std::endl;

while (rightCoxa->GetAngle() > 0) {

cv.wait(nothing, [this] {return !this->IsPause(); });

body->WholeRotate(-this->HorseStepRate / 3, body->GetRotateX(), body->GetRotateY(), body->GetRotateZ(), 1);

leftShoulder->WholeRotate(this->HorseStepRate \* 0.5, leftShoulder->GetRotateX(), leftShoulder->GetRotateY(), leftShoulder->GetRotateZ(), 1);

leftElbow->WholeRotate(this->HorseStepRate \* 3, leftElbow->GetRotateX(), leftElbow->GetRotateY(), leftElbow->GetRotateZ(), 1);

rightShoulder->WholeRotate(-this->HorseStepRate \* 0.5, rightShoulder->GetRotateX(), rightShoulder->GetRotateY(), rightShoulder->GetRotateZ(), 1);

rightElbow->WholeRotate(-this->HorseStepRate \* 3, rightElbow->GetRotateX(), rightElbow->GetRotateY(), rightElbow->GetRotateZ(), 1);

rightCoxa->WholeRotate(-this->HorseStepRate, rightCoxa->GetRotateX(), rightCoxa->GetRotateY(), rightCoxa->GetRotateZ(), 1);

leftCoxa->WholeRotate(this->HorseStepRate, 1, 0.5, -1, 1);

rightKnee->WholeRotate(this->HorseStepRate, rightKnee->GetRotateX(), rightKnee->GetRotateY(), rightKnee->GetRotateZ(), 1);

leftKnee->WholeRotate(-this->HorseStepRate, leftKnee->GetRotateX(), leftKnee->GetRotateY(), leftKnee->GetRotateZ(), 1);

float h = this->HorseStepHeight.top();

HorseStepHeight.pop();

body->Translate(0, h ,0);

this->WindowDisplay();

Sleep(20);

}

// 弥补Swing可能因为精度导致的误差，做到完全恢复状态

body->SetY(body->GetInitY());;

this->WindowDisplay();

Sleep(20);

// 两个相同的旋转可以抵消掉

leftShoulder->PopLastRotate(); //leftShoulder->PopLastRotate();

leftElbow->PopLastRotate(); //leftElbow->PopLastRotate();

rightShoulder->PopLastRotate(); //rightShoulder->PopLastRotate();

leftCoxa->PopLastRotate(); //leftCoxa->PopLastRotate();

rightCoxa->PopLastRotate(); //rightCoxa->PopLastRotate();

leftKnee->PopLastRotate(); //le/ftKnee->PopLastRotate();

rightKnee->PopLastRotate(); //rightKnee->PopLastRotate();

body->PopLastRotate(); //body->PopLastRotate();

}

自此，机器人能够从铁山靠的动作恢复到立正姿态了，且几乎没有位移。

### 3.2 7 实时暂停，旋转视角

用于机器人的动画全都在循环中，这时候不在MainLoop()中，无法对此时的按键做出响应，这就导致机器人在铁山靠的时候不能移动视角，多方位观察。要在机器人循环动作时相应按键，自然想到了多线程，将铁山靠作为一个新的线程，与主线程并行运行，这样主线程就能处理按键的输入了。并行的话需要子线程是detach。

显然，这样有一个问题就是机器人在铁山靠的同时也能往行走和重复铁山靠了。解决的方法也很简单，使用互斥锁mutex，机器人铁山靠和行走都需要锁。但是这也有一个问题：机器人在铁山靠的时候如果按了行走或者铁山靠键会导致主线程阻塞，这时候又无法处理视角变换的按键输入了。这里互斥锁对于主线程而言，不应该是阻塞，应该是跳过。最简单的方法就是在Robot中自定义一个bool变量mutex充当互斥锁，在机器人铁山靠的时候上锁，结束的时候释放锁。在上锁的时候，不对运动处理。

相关的代码如下：

RobotIkun.cpp:

case ENTER: // 回车键开始铁山靠

{

if (!robot->IsLock()) {

robot->LockMyMutex(); // 上锁

std::thread dance(&Robot::Dance, robot); // 在做铁山靠的时候能够转动视角

dance.detach(); // 主线程可以先结束的

}

glutPostRedisplay();

break;

}

Robot类：

bool mutex = 0; //不能在铁山靠的时候走路

void LockMyMutex() { mutex = 1; }

void UnlockMyMutex () { mutex = 0; }

bool IsLock() { return this->mutex; }

void Robot::Dance() {

this->TurnFrontBack(0); // 铁山靠要先背靠屏幕

this->WalkStand(); // 然后转为立正姿势

Sleep(500); // 0.5s 的准备时间

this->HorseStep(); //扎马步，抬手臂，进入准备姿势

this->Swing(); //铁山靠的摇摆

Sleep(500);

this->Reset();

this->UnlockMyMutex(); // 解锁，表明铁山靠动作做完了

}

最后是实时的暂停。暂停时需要阻塞子线程。暂停键暂停和开始交替，通过Robot.pause来表示当前状态。如果是暂停，对于行走来说，不对行走进行处理；对于铁山靠，需要在适当的位置阻塞当前线程。如果是开始，对于行走来说，对行走进行处理，对于铁山靠来说，当前线程重新开始运行。

对于行走的处理很简单，只需要if判断是否暂停决定是否调用行走方法即可。

对于铁山靠要复杂一些。这里是满足条件就阻塞线程，采用条件变量较为合适。声明该条件变量为cv，采用cv.wait()和cv.notify\_all来控制阻塞和运动。使用cv.wait()的话必需有一个unique\_lock，但我这里用自己的bool mutex代替了，所以随便声明了一个无用的unique\_lock nothing。关键在于cv.wait()第二个参数，是一个Lambda表达式，如果为True，则正常运行，否则阻塞当前线程。就通过这个表达式和pause来控制线程的状态。对于暂停，应该是随时都能暂停，这意味着在每一个while()循环的小变化中都要判断pause是否改变，如果变为1当即阻塞，达到实时暂停的效果。对于重新运行，应该是在pause = True的时候再按一次暂停键，此时调用cv.notify\_all()，通知子线程要重新运行了。

相关的代码如下：

RobotIkun.cpp:

case SPACE: //空格暂停， 开始, 通过条件变量来实现

{

std::condition\_variable cv;

robot->ChangePauseState();

robot->Notify(); // 条件变量的判断

glutPostRedisplay();

break;

}

Robot类：

bool mutex = 0; //不能在铁山靠的时候走路

bool pause = 0; //机器人暂停了

bool stopSwing= 0; //这轮铁山靠完停止

std::mutex m;

std::unique\_lock<std::mutex> nothing; // wait需要一个锁，因为这里不是传统的互斥，用变量代替这个锁了

Robot::Robot(int newX, int newY, int newZ):x(newX), y(newY), z(newZ)

{

std::unique\_lock<std::mutex> nothing1(m); // unique\_lock 得有mutex，要不然程序会崩溃

nothing = std::move(nothing1);

}

运动的代码涉及较多，这里只以TurnFrontBack()为例：

void Robot::TurnFrontBack(bool toward) {

if (this->turn <= 180) {

while (this->turn <= 180) {

cv.wait(nothing, [this] {return !this->IsPause(); });

if (toward) this->ClockwiseRotate();

else this->AntiClockwiseRotate();

this->WindowDisplay();

Sleep(10);

}

turn = 180;

}

else {

while (this->turn >= 180) {

cv.wait(nothing, [this] {return !this->IsPause(); });

// cv.wait(nothing, [this] {return !this->IsPause(); });

if (toward) this->AntiClockwiseRotate();

else this->ClockwiseRotate();

this->WindowDisplay();

Sleep(10);

}

turn = 180;

}

}

## 3.2 成果展示

由于本身有运行视频，加之该文档要打印，故以静态图片展示为主，不包含gif动图。

机器人正反外观如下图3.1，图3.2所示：

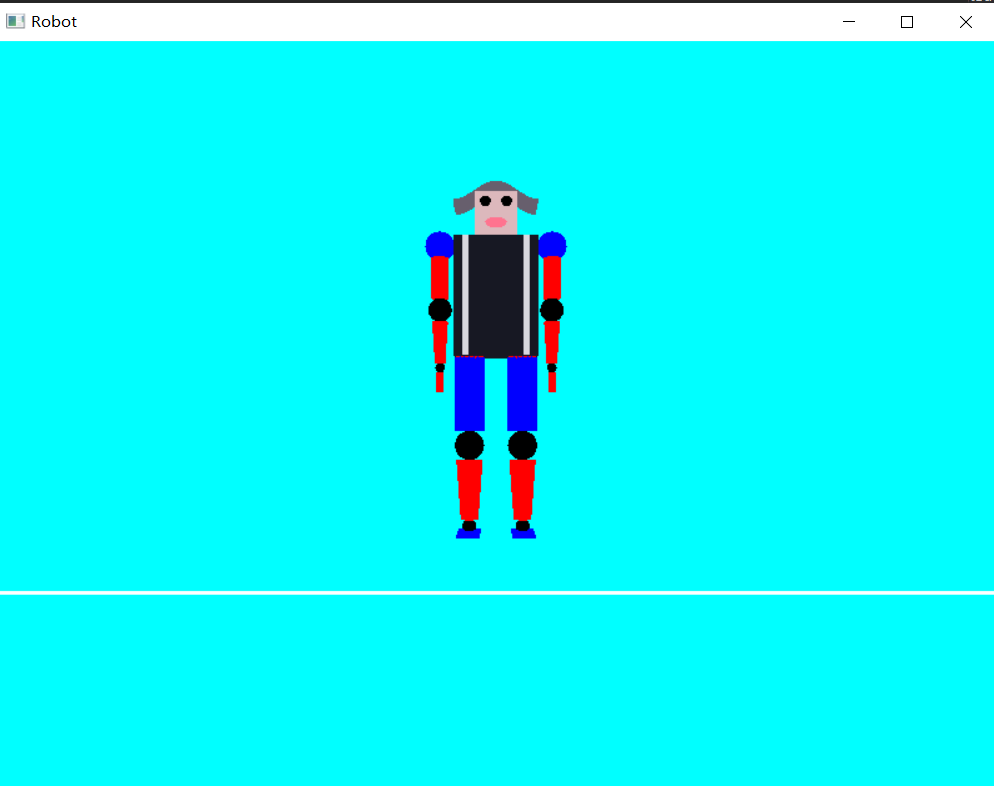


图3.1 机器人正面效果图

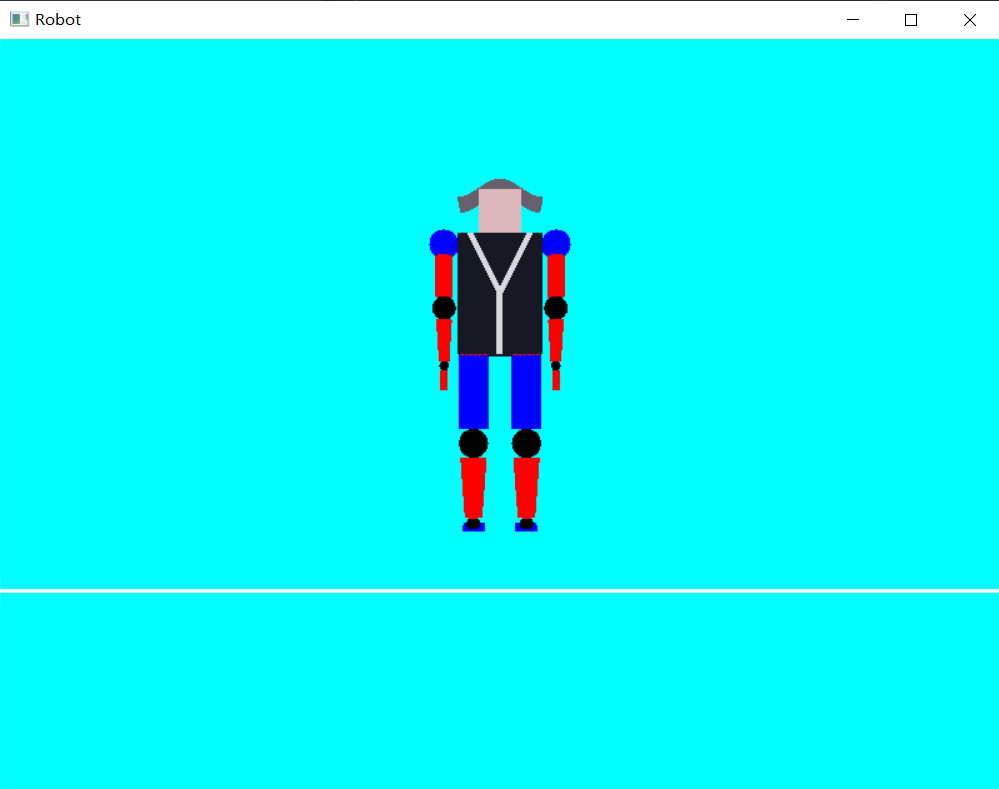


图3.2 机器人正面效果图

机器人走路姿态如下图3.3所示

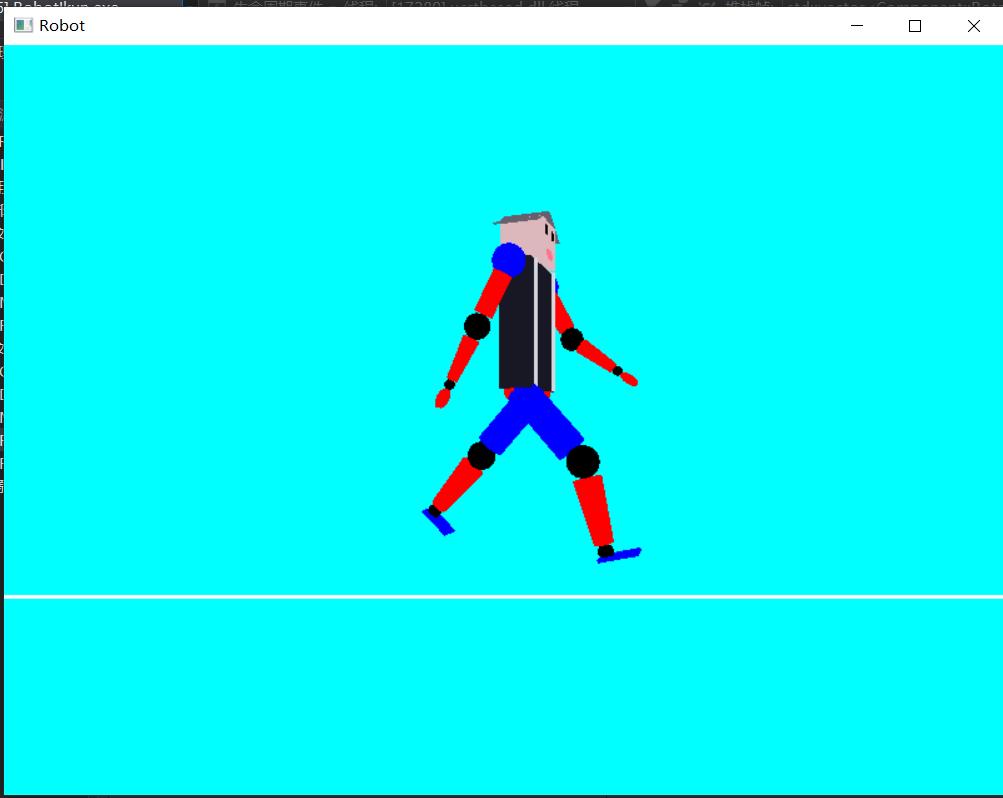


图3.3 机器人走路效果图

机器人马步，侧身如下图3.4，图3.5所示

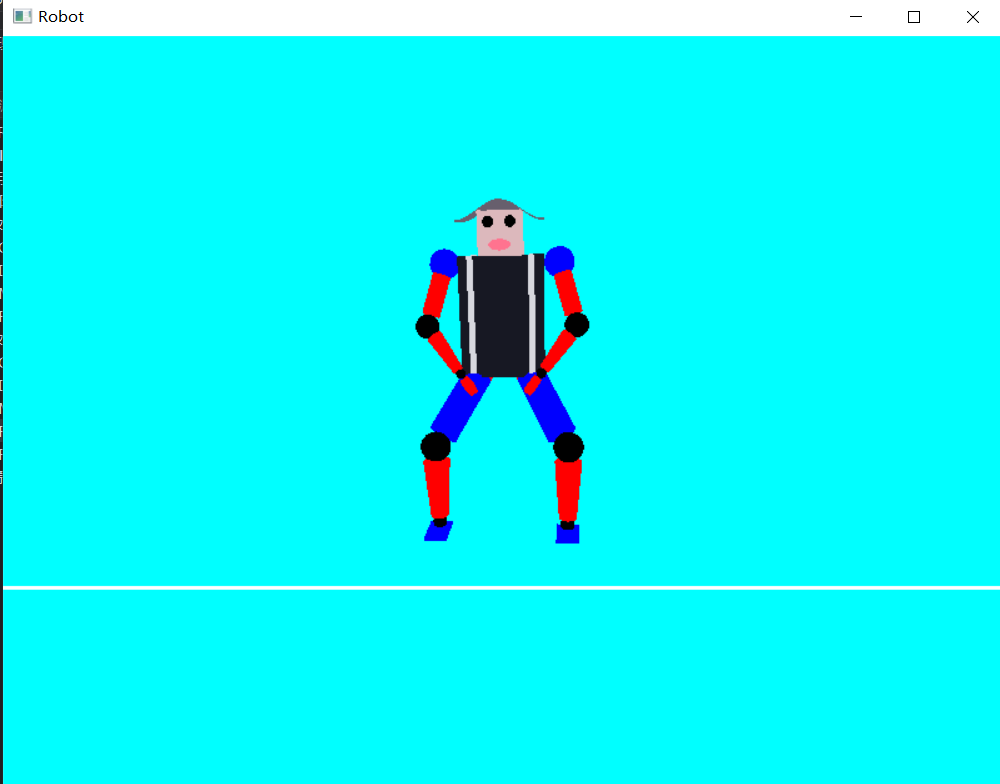


图3.4 机器人走路效果图

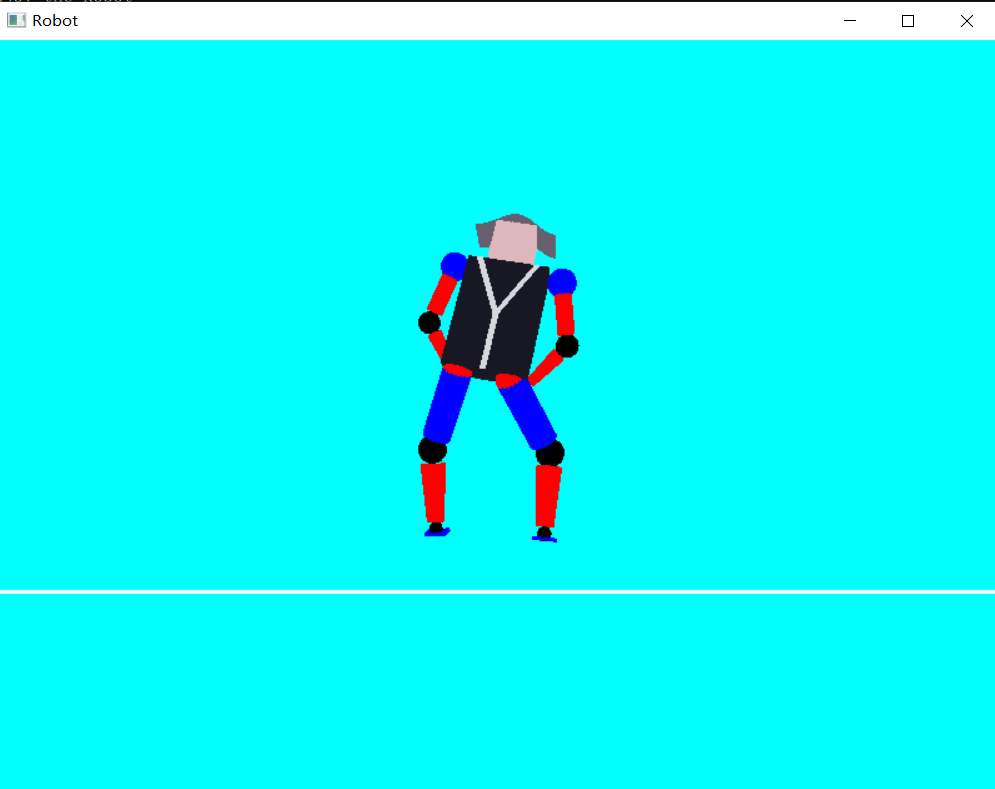


图3.5 机器人侧身效果图

机器人C型摇摆的某张定格图片如下图3.6所示

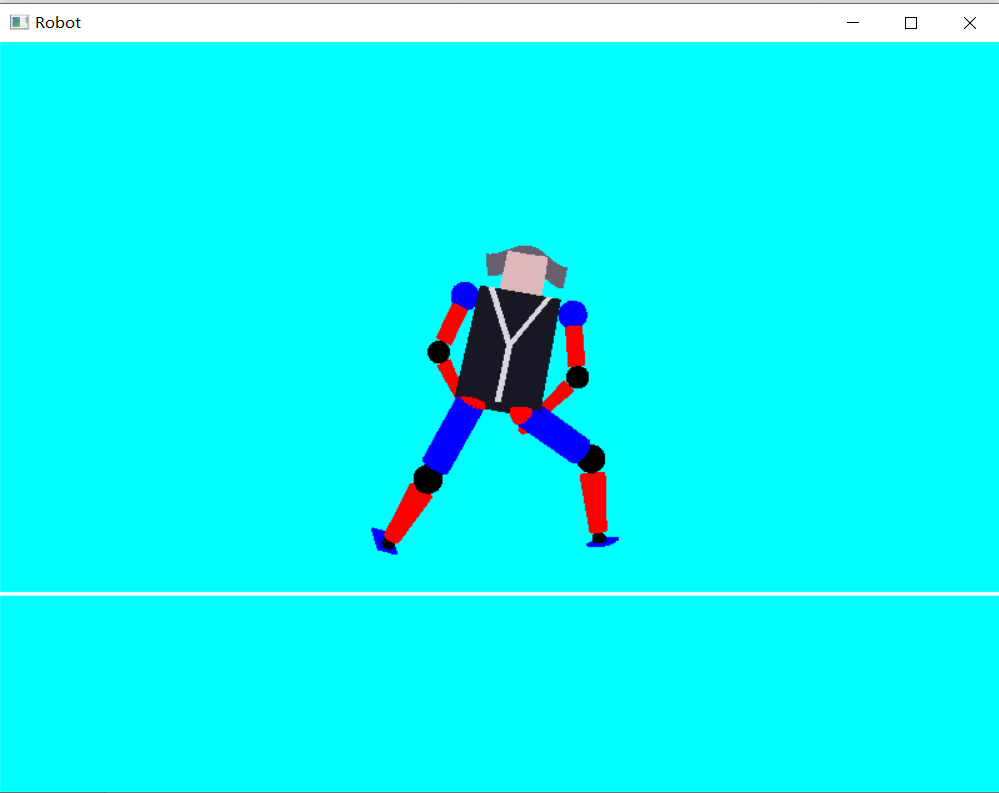


图3.6 机器人铁山靠效果图

整体成果请观看文件夹中的“RobotIkun演示视频.mp4”

# 4 项目收获

通过本次项目，我重温了C++的面向对象对象设计，同时初步实现了将opengl与面向对象相结合，使得写出的opengl程序结构不再杂乱无章。本次项目让我对opengl的ModelView有了更为深入的了解，能够运行ModelView实现一些基本的运动，并且能够系统性使用Matix，而不是面向过程那样随时随用。

通过本项目，我也了解opengl使用的一般流程，先进行初始化，然后在SetupRC()函数中设置光照，渲染等等，最后通过display等一系列相关函数绘图。

最后通过本项目，我也对C++11的特性尤其是多线程相关的有所了解。

# 附录

鉴于整体代码较长，而且分散在不同文件中，这里不附上项目代码了。将项目上传到了github个人仓库中，链接如下：

<https://github.com/gousiye/RobotIkun>